



# Head First Java


2. Classes and Objects

3. Primitives and References

## 2. Classes and Objects

- Procedural Programming: What are the **things the program has to do**? What **procedures** are needed?. OOP: What are the **things in the program**?
- Object-oriented programming lets programmer extend program without having to touch previously-tested, working code.
- All Java code is defined in a **class**. A class describes how to make an object of that class type.
- An object can take care of itself; no need to know or care how the object does it.
- When you design a class, think about the objects that will be created from that class type. Think about:
  - things the object **knows**
  - things the object **does**
- Things the object know about itself are called **instance variables** (states)

- Things the object can do are called **methods** (behaviors)
- A **class** vs. an **object**: A class is a blueprint for an object.
- Each object made from the class can have its own values for the instance variables of that class.
- The two uses of **main**
  - to **test** your real class
  - to **launch/start** your Java application
- At runtime, a Java program is nothing more than objects 'talking' to other objects.
- Garbage Collector:
  - Each time an object is created in Java, it goes into an area of memory known as **The Heap**
  - All objects - no matter when, where, or how they're created - live on the heap
  - The Java heap is actually called the **Garbage-Collectible Heap**
  - When an object is created, Java allocates memory space on the heap according to how much that particular object needs.
  - When the JVM can 'see' that an object can never be used again, that object becomes eligible for garbage collection. And if you're running low on memory, the Garbage Collector will run, throw out the unreachable objects, and free up the space, so that the space can be reused.
  - In the jar file, you can include a simple text file formatted as something called a **manifest**, that defines which class in that jar holds the `main()` method that should run

 [Back To Top](#)

### 3. Primitives and References

- Variables come in two flavors: **primitive** and **object reference**. Primitives hold fundamental values including integers, booleans, and floating point numbers. Object references hold *references* to objects.
- **VARIABLES MUST HAVE A TYPE AND A NAME**


#### Primitive Types

<u>Aa</u> Type	<u>≡</u> Bit Depth	<u>≡</u> Value Range
<u>boolean</u>	JVM-specific	true or false
<u>char</u>	16 bits	0 to 65535
<u>byte</u>	8 bits	-128 to 127
<u>short</u>	16 bits	-32768 to 32767
<u>int</u>	32 bits	-2147483648 to 2147483647
<u>long</u>	64 bits	-huge to huge
<u>float</u>	32 bits	varies
<u>double</u>	64 bits	varies

- **Rules to name variables:**
  - *It must start with a letter, underscore (\_), or dollar sign (\$). You can't start a name with a number*
  - *After the first character, you can use numbers as well. Just don't start it with a number*
  - *It can be anything you like, subject to those rules, just so long as it isn't of Java reserved words.*
- **There is actually no such thing as an object variable, there is only an object reference variable.** An **object reference variable** holds **bits** that represent a way to access an **object**. It doesn't hold the object itself, but it holds something like a pointer, or an address. JVM knows how to use the reference to get to the object.
- **Objects live in one place and one place only - the garbage collectible heap.**

- Although a **primitive variable** is full of bits representing the actual **value** of the variable, an **object reference variable** is full of bits representing a **way to get to the object**.
- *An object reference is just another variable value.*
- **Dog myDog (1) = (3) new Dog() (2)**
  - (1): Tells the JVM to allocate space for a reference variable and names the variable *myDog*
  - (2): Tells the JVM to allocate space for a new Dog object on the heap
  - (3): Assigns the new Dog to the reference variable myDog
- When you're talking about **memory allocation issues**, your Big Concern should be about **how many objects** (as opposed to object references) you're creating, and **how big they** (the objects) really are.
- All **object references** for a **given JVM** will the **same size** regardless of the size of the actual objects to which they refer, but each JVM might have a different way of representing references, so references on one JVM may be smaller or larger than references on another JVM.
- An **reference variable** has a value of **null** when it is **not referencing any object**.
- If "reference" is **the only reference** to a **particular object**, and then **it is set to null** (deprogrammed), it means that now **nobody** can get to that object it had been referring to.
- The **object reference variables** that refer to **the same object** may hold **different copies** of the same value (**reference**).
- **Arrays** are always **objects**, whether they're declared to hold primitives or object references. There is no such thing as a primitive array, only an array that holds primitives.

- It's possible to put a **byte** into an **int** array, because a **byte** will always fit into a **int-sized** variable. This is known as **implicit widening**.

 [Back To Top](#)